Representation in Evolutionary Computation

Daniel Ashlock[1], Cameron McGuinness[1], and Wendy Ashlock[2]

[1] University of Guelph, Guelph, Ontario, Canada, N1G 2W1
{dashlock,cmcguinn}@uoguelph.ca
[2] York University, Toronto, Ontario, Canada, M3J 1P3
washlock@cse.yorku.ca

The *representation* of a problem for evolutionary computation is the choice of the data structure used for solutions and the variation operators that act upon that data structure. For a difficult problem, choosing a good representation can have an enormous impact on the performance of the evolutionary computation system. To understand why this is so, one must consider the *search space* and the *fitness landscape* induced by the representation. If someone speaks of the fitness landscape of a problem, they have committed a logical error: problems do not have *a* fitness landscape. The data structure used to represent solutions for a problem in an evolutionary algorithm establishes the set of points in the search space. The topology or connectivity that joins those points is induced by the variation operators, usually crossover and mutation. Points are connected if they differ by one application of the variation operators. Assigning fitness values to each point makes this a fitness landscape. The question of the type of fitness landscape created when a representation is chosen is a very difficult one, and we will explore it in this chapter.

The primary goal of this chapter is to argue for more research into representation in evolutionary computation. The impact of representation is substantial and is not studied enough. The genetic programming community has been using *parameter sweeps* [18] which compare different choices of operations and terminals within a genetic programming environment. This is a big step in the right direction, but even this work ignores the issue of whether genetic programming is appropriate for a given problem. One of the implications of the No Free Lunch Theorem of Wolpert and Macready is that the quality of a given optimizer is problem specific. This includes the choice of representation.

There are reasons that representation has not been explored. While there can be huge rewards from exploring different representations, there is also a substantial cost. One must implement alternate representations; one must run well-designed experiments with them which probably include parameter tuning for each representation; and then one must find a way to compare the solutions. This last task seems simple – could not one simply examine final fitness numbers? While the first answer to this question is clearly yes, it may be that a problem requires diverse solutions or robust solutions. The recent explosion of research in multicriteria optimization with evolutionary algorithms means that issues like the diversity of solutions produced are important.

We will examine the question of representation through a series of examples involving: a simple toy optimization problem, the problem of evolving game playing agents, real optimization problems, and, finally, a problem drawn from automatic content generation for games.

1 Representation in Self-avoiding Walks

The *self-avoiding walk* (SAW) problem involves traversing a grid, given instructions for each move, in such a way that every square is visited. Fitness is evaluated by starting in the lower left corner of the grid and then making the moves specified by the chromosome. The sequence of moves made is referred to as the *walk*. If a move is made that would cause the walk to leave the grid, then that move is ignored. The walk can also revisit cells of the grid. Fitness is equal to the number of squares visited at least once when the walk is completed. The problem is called the *self-avoiding* walk problem because optimal solutions for a number of moves equal to the number of squares minus one do not revisit squares; they are self-avoiding walks. Figure 1 shows the 52 global optima for the 4×4 SAW problem. In addition to a diverse set of optimal solutions, the SAW problem has many local optima when the grid is large enough.

The SAW problems has a number of nice qualities as an evolutionary computation test problem:

- The problem has a large number of cases, one for each possible size of grid. While problem difficulty does increase with grid size, it is also different for grids of the same size with different dimensions such as 4×4 and 2×8.
- Even for quite long genes, the solutions have a simple two-dimensional representation. This makes visualizing final populations easy. Visualizations of the final walk also make it easy to compare between different representations.
- The problem, when the grid is large enough, has a large number of both global and non-global optima. This starts, roughly, when both dimensions are larger than 3. Table 1 gives the number of global optima.
- The global optima are not symmetrically distributed. Some have many other optima nearby, while others are far from other optima. This means that, even though they have the same fitness, they differ in how easy they are to locate. The notion of *nearby* used here is Hamming distance.

Having made a case that the SAW has desirable properties for a test problem, the next step is to construct multiple representations for it. We will examine three representations, one of them the obvious choice, and all implemented as strings over some alphabet. Other than changing the representation, all experiments will be performed using a population of 100 strings using two-point crossover and a mutation operator that changes two of the characters in the string. The problem case used is the 4×4 SAW.

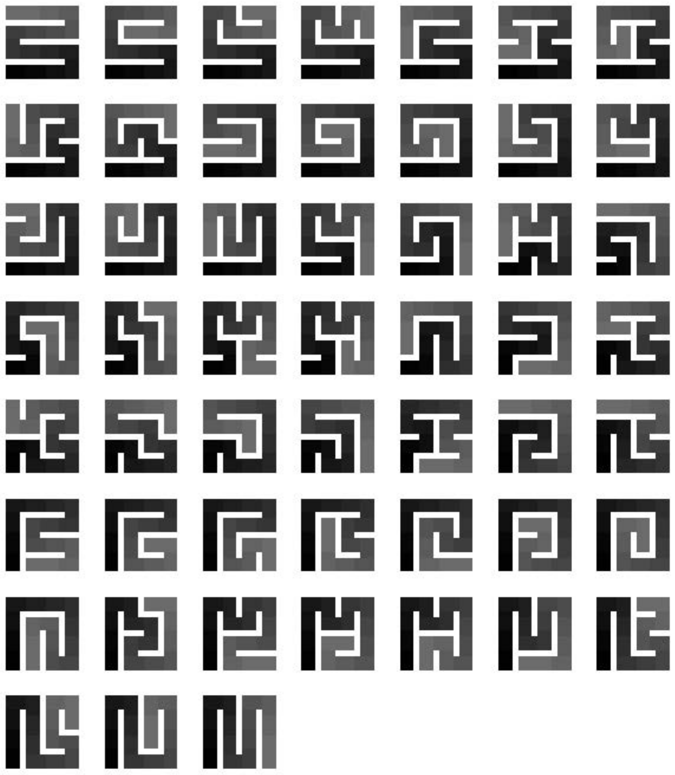

Fig. 1. The optimal solutions to the 4 × 4 SAW problem

The Direct Representation

The direct representation uses a character string over the alphabet $\{U, D, L, R\}$, which stand for *Up*, *Down*, *Left*, and *Right*. The string is of length fifteen and the fitness function simply executes the moves in order, recording the number of squares visited. Since evaluation starts in the lower left square with that square already visited the minimum fitness is one and the maximum is 16. The string length is equal to the minimum number of moves required to visit all the squares.

The Relative Representation

The relative representation uses a character string of length 15 over the alphabet $\{F, R, L\}$ which stand for *forward*, *turn right and then move forward*, and *turn left and then move forward*. Like the direct representation, the relative representation keeps track of the square it currently occupies. It adds to that information the direction it is currently facing. Fitness evaluation starts with the drawing agent facing upward. Fitness evaluation is otherwise like the direct representation.

Table 1. Number of global optima in the SAW problem for problem sizes $2 \leq n, m \leq 7$

n/m	2	3	4	5	6	7
2	2	3	4	5	6	7
3	3	8	17	38	78	164
4	4	17	52	160	469	1337
5	5	38	160	824	3501	16,262
6	6	78	469	3501	22,144	144,476
7	7	164	1337	16,262	14,4476	1,510,446

The Gene Expression Representation

The gene expression representation uses a character string over an alphabet derived from the one used in the direct representation: $\{U, D, L, R, u, d, l, r\}$. During fitness evaluation upper case letters are used normally and lower case letters are ignored. If a gene has fewer than fifteen upper case letters, fitness evaluation simply ends early, an implicit fitness penalty. If a gene has more than fifteen upper case letters, only the first fifteen are used. In order to permit the average number of upper case letters to be fifteen, the length of the string is set to 30. The name of the gene expression representation reflects that the upper/lower case status of a character controls the *expression* of each gene loci.

Results

A simple assessment of the impact of changing the representation is given in Figure 2. The time to solution for sets of 1000 replicates done for all three representations was sorted and then graphed. The performance of the representations is strongly stratified with the direct representation exhibiting the worse performance (longer times to solution), the gene expression representation coming in second, and the relative representation coming in first. For the replicates with the longest time to solution (right end of the sorting order), the gene expression representation takes over for first place.

The goal of demonstrating that the choice of representation makes a difference has been met for the SAW problem. Let us now consider what caused the change in performance. The size of the search space for the relative representation is 3^n, while for the direct representation the size is 4^n, meaning that evolution has a smaller job to do. The relative representation encodes far fewer walks than the direct one. In particular, the relative representation is incapable of moving back to the square it just came from, a move that always results in a sub-optimal solution. This gives the relative representation a substantial advantage: it retains all the optimal solutions in the direct representation while excluding many sub-optimal ones. This is an example of building domain information into the representation.

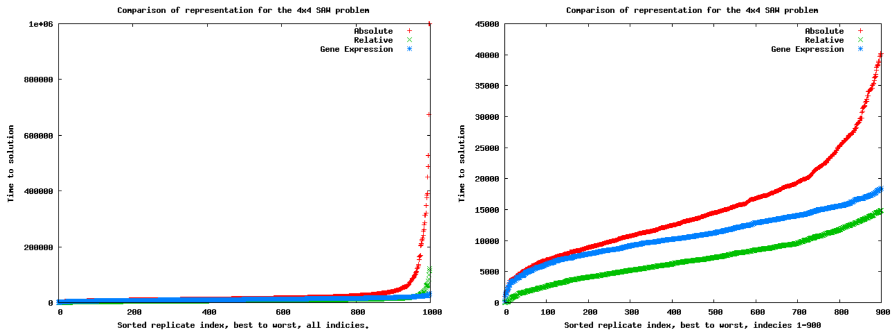

Fig. 2. This figure shows the impact of changing representation on the time to solution for the 4x4 SAW problem. The graphs display the sorted times to solution for 1000 independent evolutionary replicates. The left panel displays all 1000 replicates while the right one displays only the first 900.

The best average performer is the gene expression representation, though this is due to a small number of bad results for the relative representation. The size of the search space here is 8^{30}, enormously larger than direct or relative representation. This demonstrates that the size of the search space, while potentially relevant, cannot possibly tell the whole story. Both the direct and relative representation uniquely specify a sequence of moves. The gene expression representation has billions of different strings that yield the same sequence of moves. It also specifies some sequences of moves the other two representations cannot, but these all contain fewer than fifteen moves and have intrinsically bad fitness.

To understand the good performance of the gene expression representation, it is necessary to consider the fitness landscape. A mutation of a gene in the direct or relative representation changes one move in the walk represented by that gene. Some of the mutations in the gene expression representation have the same effect, but those that change a capital letter into lower case or vice versa have the effect of inserting or deleting characters from the walk specified by the gene. This means that the gene expression representation has *edit* mutations that can insert, delete, or change the identity of a character in the walk the gene codes for. The other two representations can only change the identity of characters.

If we consider the space of encoded walks, rather than genes, the gene expression representation has more connectivity. If we think of optima as hills in the fitness landscape, then using the gene expression representation has the effect of merging some of the hills. Since the number of optimal results remains constant, this means the only effect is to eliminate local optima.

One weakness of this demonstration of the impact of representation on the SAW problem is that only one case of the problem was examined. Figure 3 shows the result of performing the same experiments for the 5×5 case of the SAW problem. The algorithm was set to halt if it did not find a solution in 1,000,000 fitness evaluations. This is what causes the flat portions of the plots on the right side of the figure. Notice that, in this experiment, the order of the direct and relative representations is reversed and the gene expression representation is clearly the best.

This second example serves to demonstrate that the representation issue is complex, even on a problem as simple as the SAW. Another quality of the SAW, demonstrated in [15], is that different cases of the problem behave differently from one another as optimization problems. The results in Figure 3 provide additional evidence that different sizes of SAW problems are substantially different from one another.

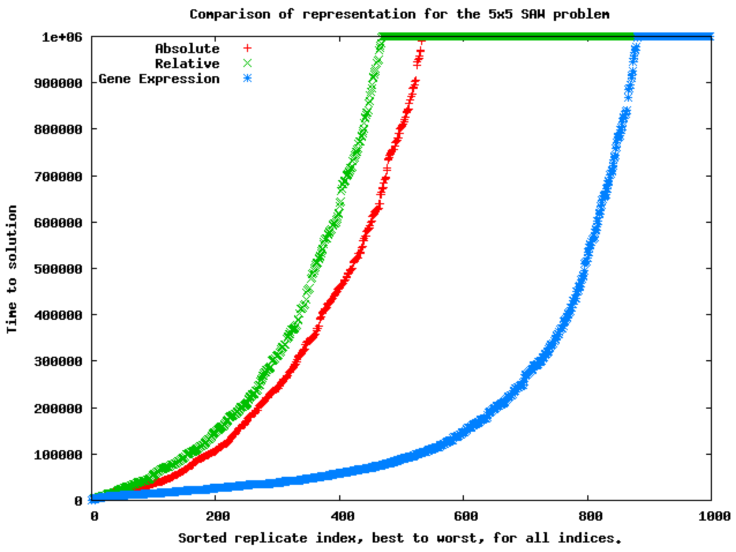

Fig. 3. This figure shows the impact of changing representation on the time to solution for the 5x5 SAW problem. The graphs display the sorted times to solution for 1000 independent evolutionary replicates.

2 Representation in Game-Playing Agents

The game used to demonstrate the impact of representation on the evolution of game playing agents is the iterated prisoner's dilemma. The *prisoner's dilemma* [13] is a widely known abstraction of the tension between cooperation and conflict. In the prisoner's dilemma two agents each decide simultaneously, without communication, whether to cooperate (C) or defect (D). If both players cooperate, they receive a payoff of C; if both defect, they receive a payoff of D. If one cooperates and the other defects, then the defector receives the *temptation* payoff T, while the cooperator receives the *sucker* payoff S. In order for a simultaneous two-player game to be prisoner's dilemma two conditions must hold:

$$\mathcal{S} \leq \mathcal{C} \leq \mathcal{D} \leq \mathcal{T} \tag{1}$$

and

$$(\mathcal{S} + \mathcal{T}) \leq 2\mathcal{C} \tag{2}$$

The first of these simply places the payoffs in their intuitive order while the second requires that the average score for both player's in a unilateral defection be no better than mutual cooperation.

A situation modeled by the prisoner's dilemma is that of a drug dealer and an addict exchanging money for drugs in simultaneous blind drops to avoid being seen together by the police. Cooperation consists of actually leaving the money or drugs; defection consists of leaving something worthless like an envelope full of newspaper clippings in place of money or an inert white powder in place of the drugs. If the exchange is made only once, then neither party has an incentive to do anything but defect. If the drop is to be made weekly, into the indefinite future, then the desire to get drugs or money next week strongly encourages cooperation today. This latter situation is an example of the *iterated* prisoner's dilemma. When play continues, the potential for future retribution opens the door to current cooperation. The payoff values used in the experiments described here are $S = 0$, $D = 1$, $C = 3$, and $T = 5$, largely because these values have been used in many other studies in which prisoner's dilemma agents were evolved [2,10,4,9].

Earlier research [5,1] compared ten different representations for the iterated prisoner's dilemma. These experiments all used populations of 36 agents whose fitness was computed as the average score in a round-robin tournament of 150 rounds of iterated prisoner's dilemma between each pair of players. Each agent had access to their opponent's last three plays, and perhaps more in the case of state conditioned representations. Evolution was run for 250 generations with the crossover and mutation operators kept as similar as possible given the differing representations.

Representations

The representations studied for the iterated prisoner's dilemma are as follows:

Finite State Machines. Two types of finite state machines are used: directly encoded finite state machines with 16 states (**AUT**) and finite state machines represented with a developmental encoding [19]. The number of states in the machine is variable but not more than twenty. These finite state machines are referred to by the tag **CAT**.

Function Stacks. The tags **F40**, **F20**, and **F10** are for *function stacks*, a linear genetic programming representation based on a directed acyclic graph. The data type is boolean and the operations available are logical And, Or, Nand, and Nor. The constants true and false are available as are the opponent's last three actions.

We use the encoding true=defect, false=cooperate. The numbers 10, 20, and 40 refer to the number of nodes in the directed acyclic graph.

Tree-Based Genetic Programming. We use standard tree-based genetic programming [20] with the same encoding as the function stacks and the same Boolean functions with access to the opponent's last three actions. These are referred to with the tag **TRE**. The tag **DEL** corresponds to Boolean parse trees, identical to TRE,save that a one-time-step delay operator is incorporated into the operation set.

Markov Chains. The tag **MKV** is used for Markov chains implemented as look-up tables indexed by the opponent's last three actions that gives the probability of cooperation. Once this probability has been found a random number is used to determine the agent's action. The tag **LKT** is used for look-up tables indexed by the opponent's last three actions. The lookup tables are like the Markov chains if the only probabilities permitted are 0 and 1.

ISAc Lists. A different linear genetic programming representation denoted by **ISC** are If-Skip-Action lists [12]. An ISAc list executes a circular list of Boolean tests on data items consisting of the opponent's last three actions and the constants "cooperate" and "defect" until a test is true. Each Boolean test has an action associated with it, the action for the true test is the agent's next action. On the next round of the game execution starts with the next test. The lists of tests used here have a length of 30.

Neural Nets. The tag **CNN** is used for feed-forward neural nets with a per-neuron bias in favor of the output signifying cooperation; they access the opponent's last three actions and have a single hidden layer containing three neurons. The tag **NNN** are feed-forward neural nets identical to CNN save that they have no bias in favor of cooperation or defection.

2.1 Results

The metric used to compare representations is the probability the final population, at generation 250, is essentially cooperative. We measure this as having an average payoff of 2.8 or more. This is a somewhat arbitrary measure, carefully justified only for the **Aut** representation. For finite state automata, a series of initial plays between two players must be followed by a repeating sequence of plays caused by having reached a closed loop in the (finite) space of states. When fitness evaluation consists of 150 rounds of iterated prisoner's dilemma and the automata have no more than sixteen states, an average score of 2.8 or more corresponds to having no defections in the looped portion of play.

Figure 4 shows the probability that different representations will be cooperative. This result is, in a sense, appalling. The outcome of the basic experiment to demonstrate that cooperation arises [22] has an outcome that can be dialed

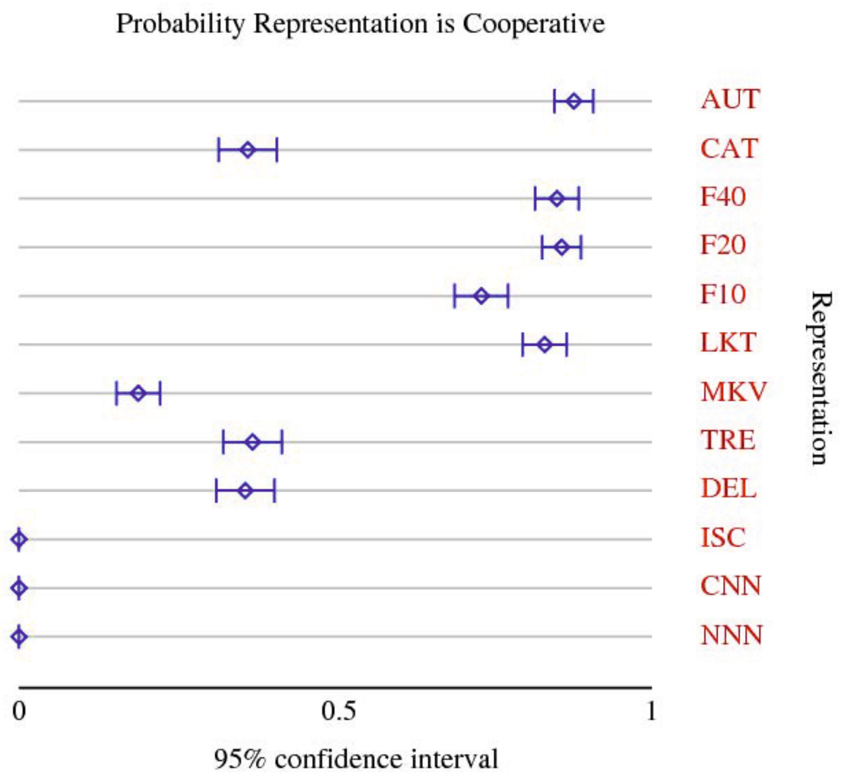

Fig. 4. Shown are 95% confidence intervals on the probability that the final generation of an evolutionary algorithm training prisoner's dilemma playing agents, with different representations, will be cooperative

from 95% cooperative to no cooperation by changing the representation. This shows not only that representation has an impact but that it can be the dominant factor. In other words, an experiment using competing agents that does not control for the effects of representation may have results entirely dictated by the choice of representation.

There are a number of features of these experiments that make the situation worse. In order to check the importance of changing the parameters of a single representation, the function stack (Boolean directed acyclic graph genetic programming) representation was run with 10, 20, and 40 nodes. Notice that the 10-node version of this representation is significantly less cooperative than the others.

In a similar vein, the **AUT** and **CAT** are alternate encodings of the same data structure. In spite of this they have huge differences in their degree of cooperativeness. There is a mathematical proof in [1] that the function stack representation encodes the same space of strategies as finite state machines.

A slightly different version of this fact applies to the two neural net (**CNN** and **NNN**) and the lookup table (**LKT**) representations. All of these are a map from the opponent's last three actions to a deterministic choice of an action. It is not difficult to show all three representation encode *exactly* the same space of 2^8 strategies in different ways. In spite of this the neural net representations have an experimentally estimated probability of zero of cooperating; the lookup tables are among the most cooperative representations.

3 Representation in Real Optimization

Real parameter optimization is one of the earliest applications of evolutionary computation. Evolution strategies [14] were originally designed to optimize parameters that described an airfoil and also has had substantial success at designing nozzles that convert hot water into steam efficiently. Real parameter optimization also substantially pre-dates evolutionary computation; it is one of the original applications of the differential calculus with roots in the geometry of the third century B.C. and modern treatments credited to Isaac Newton and Gottfried Leibniz in the seventeenth century. The natural representation from the calculus, as functions mapping m-tuples of numbers to a single parameter to be optimized, is a natural one only adopted by some techniques within evolutionary computation.

The largest difference between real optimization and representations built on character strings is the set of available mutation operators. When we change the value of a real number, that change is a probability distribution on the real numbers. It could be uniform, Gaussian, or some more exotic distribution. Evolutionary programming [16] pioneered the use of mutation operators that use covariance across parameters to permit evolution to modify mutation operators to respect the local search gradient. One of the representations we will examine completely avoids the issue of selecting the correct distribution for a mutation operator, while the other two retain their critical dependence on that choice. Correct choice of the type of mutation operator in real parameter optimization is very important, but it is not the subject we are concerned with in this article.

3.1 Representations

We will examine three possible representations for real-parameter optimization. There are many others. Some of the earliest work in real optimization [17] represented sets of real parameters as strings of bits with blocks of bits first mapped onto an integer value and then the integer value used to pick out a value from an equally spaced set for a given parameter value. This representation required techniques, like Grey-coding, to ensure that some bits were not far more important than others. This is a nice, early example of finding a more effective representation.

The Standard Direct Representation. Current evolutionary real optimization often operate on vectors of real numbers holding the parameters to be optimized in some order. We will call treating such a vector of real numbers as a string of values, using crossover operators analogous to the string like ones, the *standard direct representation.* This is the first of our three representations.

The Gene Expression Representation for Real Parameters. The gene expression representation, used on the SAW problem in Section 1, can easily be adapted to real parameter optimization. We first lengthen the vector of real parameters by double and then add an *expression layer* in the form of a binary gene with one loci for each parameter in the vector of reals. Before the vector of reals is sent to the fitness function, an *expression step* is performed. Suppose that n real parameters are required. Only those real values with a one in the corresponding position in the expression layer are used. If fewer than n real parameters are expressed in this fashion, then the individual receives fitness that is the worst possible. If n or more parameters are expressed, then the first n, in the order they appear in the data structure, are used. In this case, the usual fitness for those n parameters is the fitness of the entire data structure.

The Sierpinski Representation. The *Sierpinski representation* first appears in [11] and was used in [3] to located parameters for interesting subsets of the Mandelbrot set. The Sierpinski representation is inspired by the *chaos game*, an iterative averaging algorithm for generating the Sierpinski triangle or gasket, shown in Figure 5. The game starts at any vertex of the triangle. The game then iteratively moves half way toward a vertex of the triangle selected uniformly at random and plots a point. The points have been colored by averaging a color associated with each vertex into a color register each time a particular corner was selected. This visualizes the importance of each vertex to each plotted point.

If, instead of the three vertices of a triangle, we use the 2^n points that are the vertices of an n-dimensional box, then a series of averaging moves toward these points specify a collection of points that densely covers the interior of the box [11]. Strings of averaging moves form the representation for evolutionary search. Formally:

Definition 1. The Sierpinski representation. *Let* $G = \{g_0, g_1, \ldots, g_{k-1}\}$ *be a set of points in* $\mathbb{R}^n$ *called the* generator points *for the Sierpinski representation. Typically these are the vertices of a box in some number of dimensions. Associate each of the points, in order, with the alphabet* $\mathcal{A} = \{0, 1, \ldots, k-1\}$*. Let the positive integer* r *be the* depth of representation*. Then for* $s \in \mathcal{A}^r$ *the* point represented by s*,* p_s*, is given by Algorithm 31.*

Definition 2. *The* **normalized Sierpinski representation***(NSR) is achieved by insisting that the last character of the string always be the first generator.*

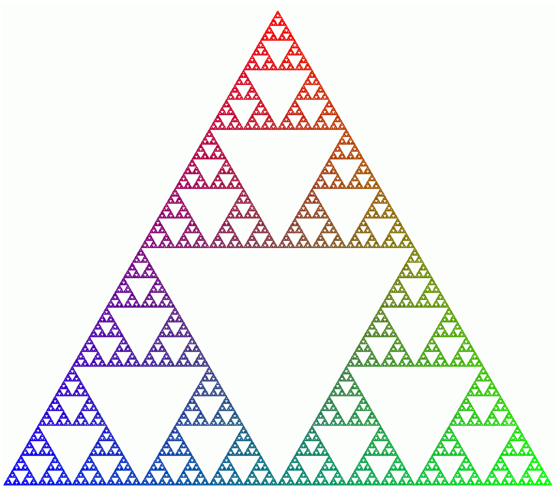

Fig. 5. The Sierpinski triangle or gasket

Algorithm 31. Sierpinski Unpacking Algorithm

```
Input: A string $s \in \mathcal{A}^r$;The set $G$ of $k$ generator points;
       An averaging weight $\alpha$
Output: A point in $\mathbb{R}^n$
Details:
Set $x \leftarrow g_{s[r-1]}$
for($i \leftarrow r-2$; $i \geq 0$; $i \leftarrow i-1$)
    $x \leftarrow \alpha \cdot g_{s[i]} + (1-\alpha) \cdot x$
end for
return($x$)
```

The following lemma is offered without proof (but is elementary).

Lemma 1. *Let a string s of length r be a* name *for a point $x = p_s$. Suppose that $\alpha = 0.5$ and that the last character (initial generator) of s is always the same. Then s is the sole name of x of length r.*

The Sierpinski representation reduces the problem of real optimization to that of evolving a string. Lemma 1 tells us that each string in the normalized version of the representation corresponds to a unique point. An important feature of the Sierpinski representation is that it searches only inside the convex hull of the generators. This has good and bad points; the generators can be used to direct search, but the search *cannot* use a mutation operator to locate an optima outside of the initial boundaries in which the population was initialized – something both the other representations can do.

3.2 Comparing the Direct and Sierpinski Representations

We will compare the direct and Sierpinski representations on the problem of optimizing the function:

$$h(x_0, x_1, \ldots, x_{n-1}) = sin(\sqrt{x_0^2 + x_1^2 + \cdots + x_{n-1}^2}) \cdot \prod sin(x_i) \tag{3}$$

in $n = 5$ dimensions. This function possesses an infinite number of optima of varying heights and is thus good for testing the ability of an algorithm to locate a diversity of optima.

Notice that the Sierpinski representation stores points as strings of characters. This means that we can store and retrieve points in a dictionary – with logarithmic time for access – and can compare points for "nearness" by simply checking their maximum common prefix. In particular, if we are searching a space with multiple optima, it becomes very easy to database optima that the algorithm has already located. The MOSS, given as Algorithm 32, was first specified in [11].

Algorithm 32. Multiple Optima Sierpinski Searcher (MOSS)

```
Input: A set of generator points $G$
       An averaging parameter $\alpha$
       A depth of representation $r$
       A depth of exclusion $d$
       A multi-modal function $f$ to optimize
Output: A collection of optima
Details:
Initialize a population of Sierpinski representation strings
Run a string-EA until an optimum $x$ is found
Initialize a dictionary $D$ with the string specifying $x$
Repeat
  Re-run the EA, awarding minimal fitness to any string
      with the same $d$-prefix as any string in the dictionary
  Record the new optimum's string in $D$
Until(Enough optima are found)
```

The MOSS algorithm creates zones of low fitness around the optima it has already located. The size of the zones is determined by the exclusion depth and have a shape identical to the convex hull of the generators. Each increase in the exclusion depth decreases the size of the holes around known optima by one-half.

Table 2 compares 100 runs of the standard algorithm with 100 runs of the MOSS algorithm supported by the Sierpinski representation. The goal, in this case, is to locate as many optima as possible. The table gives a tabulation of optima located stratified by the number of times they were located. The results are striking: the MOSS algorithm located far more optima (969) than the standard

Table 2. Relative rate of location among the 1000 populations optimizing Equation 3 for the original and MOSS algorithms

Times	Number of Optima		Times	Number of Optima	
located	Original	MOSS	located	Original	MOSS
1	122	938	8	6	0
2	77	31	9	1	0
3	56	0	10	3	0
4	40	0	11	1	0
5	19	0	12	1	0
6	15	0	13	1	0
7	14	0			

algorithm (356) and never located a given optima three times. The standard algorithm located six optima more than ten times each. The average quality of the optima located is higher for the standard algorithm, because it locates high quality optima multiple times. The two representations compared are not, in an absolute sense, better or worse. Rather, each has situations in which it is better. The strength of the Sierpinski representation is locating a diversity of optima; it makes databasing optima easy and so enables the MOSS algorithm.

3.3 Comparison of the Direct and Gene Expression Representations

We compare the standard direct and gene expression representations on the function:

$$g_n(x_1, x_2, \ldots, x_n) = \frac{1}{20n} \sum_{k=0}^{n} x_k + \sum_{k=0}^{n} sin(\sqrt{k} \cdot x_k) \tag{4}$$

in two through seven dimensions. This problems has many local optima and the small linear trend means that the better optima are further afield.

For each dimension and representation, 400 replicates of an evolutionary algorithm were run and a 95% confidence interval on the quality of the optima located were constructed. This confidence interval was constructed at both 100,000 fitness evaluations and 1,000,000 fitness evaluations. The results are given in Table 3.

The advantage of using the gene expression representation is largest in lower dimensions. It ceases to be significant when we compare the results in $d = 7$ dimensions for the shorter evolutionary runs. The significance returns in the longer evolutionary runs. This demonstrates that the gene expression representation made better use of additional time.

The fitness landscape for this problem is easy to understand - it has a lot of hills and the small linear trend means that searching further afield will always locate better optima. This lets us draw the following conclusion: the gene

Table 3. Mean value of best optima located, averaged over replicates, and best optimum located in any replicate for the polymodal function. This table compares the standard direct representation and the gene expression representations for two different lengths of evolution.

	100,000 fitness evaluations				**1,000,000 fitness evaluations**			
	Gene Expression		Direct		Gene Expression		Direct	
Dimension	Mean	Best	Mean	Best	Mean	Best	Mean	Best
2	2.77 ± 0.02	3.25	2.59 ± 0.02	2.98	3.04 ± 0.03	4.17	2.58 ± 0.02	2.98
3	3.74 ± 0.01	4.11	3.54 ± 0.02	3.97	3.94 ± 0.02	4.45	3.55 ± 0.02	3.97
4	4.71 ± 0.01	5.05	4.54 ± 0.01	4.97	4.88 ± 0.01	5.33	4.52 ± 0.01	4.97
5	5.68 ± 0.02	5.98	5.51 ± 0.01	5.85	5.80 ± 0.02	6.31	5.52 ± 0.01	5.87
6	6.58 ± 0.02	6.95	6.49 ± 0.01	6.82	6.73 ± 0.01	7.10	6.51 ± 0.01	6.90
7	7.54 ± 0.04	7.93	7.50 ± 0.01	7.85	7.68 ± 0.02	8.08	7.50 ± 0.01	7.80

expression representation is better at exploration, while the standard direct representation is better at exploitation. It is easy to test this hypothesis in the opposite direction by optimizing a different function. Recall that mutations to the binary expression layer amount to inserting or deleting values from the sequence of real parameters. The gene expression representation preforms badly when optimizing a unimodal function with its mode selected so that no two of its coordinates are the same (data not shown). It has a far worse mean time to solution than the direct representation in low dimensions and completely fails to locate the optimum in higher dimensions.

As with the Sierpinski representation, the gene expression representation is neither better nor worse than the standard direct one. Each has its own appropriate domain of applicability. This is additional support for the thesis of this chapter, that we should study representation more assiduously. Each new representation is an additional item in our toolbox. Both the comparisons made in this section demonstrate that there is a significant impact to the choice of representation.

4 Representation in Automatic Content Generations

In this section we look at the problem of evolving a maze. Examples of the types of mazes we are evolving are shown in Figures 8 and 9. We will use the same evolutionary algorithm for each of five representations. The representations used in this study are defined in [6,8,21]. Since none of the cited publications used the same fitness function on all five representations we use a new fitness function. The mazes we are evolving are specified on a grid. The mazes have an entrance in the center of each wall and two internal checkpoints. Figure 7 designates long and short distances. These distances are the lengths of the shortest paths between

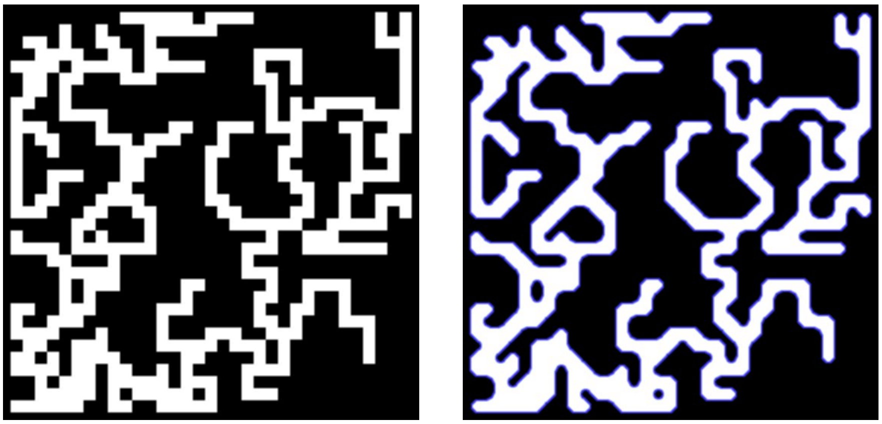

Fig. 6. Shown are a raw (left) and rendered (right) maze of the sort specified by evolving which squares on a grid are obstructed

the specified points. The fitness function is the sum of the long distances divided by the sum of the short distances, except that any maze where we cannot move from each door to both checkpoints is awarded a fitness of zero. Distances are computed using a simple dynamic programming algorithm [6].

We will demonstrate that the visual character of the mazes changes substantially when the representation is changed. All the evolutionary algorithms use a population of 100 mazes stored as strings of values with a 1% mutation rate and two-point crossover. Evolution proceeds for 500,000 fitness evaluations. The representations are as follows:

First Direct Representation. Open and blocked squares within a rectangular grid are specified directly as a long, binary gene.

Chromatic Representation. A direct representation, in which the squares within a grid are assigned colors from the set { red, orange, yellow, green, blue, violet}. These colors are specified directly as a long gene over the alphabet $\{R, O, Y, G, B, V\}$. An agent can move between adjacent squares if they are (i) the same color or (ii) adjacent in the above ordering.

Height-Based Representation. A direct representation, in which the squares within a grid are assigned heights in the range $0 \leq h \leq 10.0$. An agent can move between adjacent squares if their heights differ by 1.0 or less.

Indirect positive representation. The chromosome specifies walls that are placed on an empty grid to form the maze. The walls can be horizontal, vertical, or diagonal. In this representation walls are explicit, and rooms and corridors are implicit.

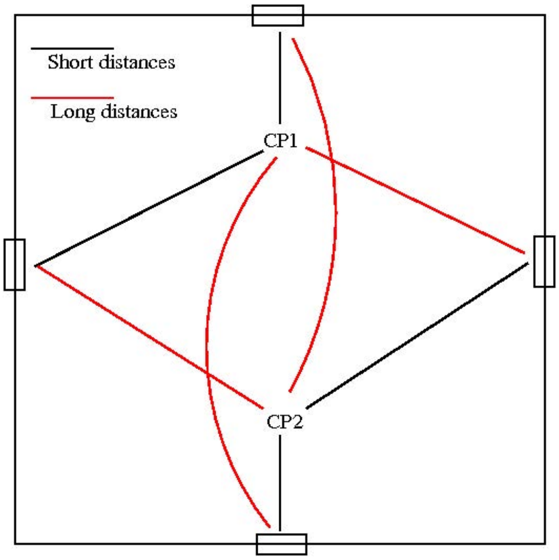

Fig. 7. This figure shows the four doors and internal checkpoints within a maze. The fitness function maximizes the quotient of the sum of the red distances and the sum of the black distances. This permits the evolution of a diverse collection of mazes with similar properties.

Indirect Negative Representation. The chromosome specifies material to remove from a filled grid to form the maze. In this representation, rooms and corridors are explicit, and walls and barriers are implicit.

All of the representations, except the indirect negative representation, use a technique called *sparse initialization* to compensate for the fact that, when the data structure is filled in uniformly at random, it is quite likely to have zero fitness, because there is no path between at least one door and at least one checkpoint. Sparse initialization biases the initial population to have high connectivity. Sparse initialization takes the following forms. For the direct representation, only 5% of the squares are filled in. For the chromatic representation all squares are initialized to green or yellow. For the height-based representation the heights are initialized to a Gaussian random value with mean three and height one. For the indirect positive representation all walls start at length three. Using sparse initialization places the burden of building the maze onto the variation operators. The initialization to a highly connected state biases the trajectory of evolution.

4.1 Results

Figure 8 gives examples of evolved mazes for the direct and indirect negative and positive representations. Figure 9 gives the examples for the chromatic and height representations. Since it is very hard to see paths in these mazes, they are accompanied by a key where non-adjacent squares are separated by walls and inaccessible squares are blacked out.

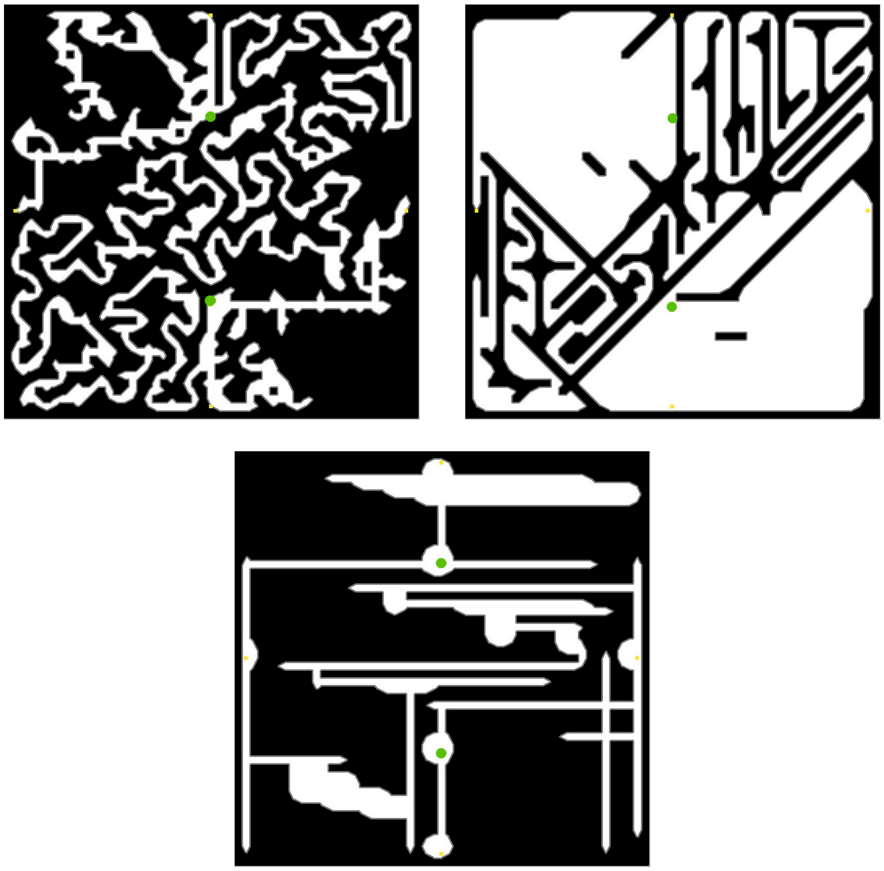

Fig. 8. Examples of the direct, positive, and negative representations for making mazes. The checkpoints are shown as green circles.

The results in this section speak for themselves. Even though they are evolved to satisfy the same distance-based fitness function the overall appearance of the mazes is very different. The appearance is entirely dependent on the choice of representation. The two most similar representations are the chromatic and height based. The keys to these mazes look similar. The actual mazes, though, look quite different. The type of representation that should be chosen, in this example, depends strongly on the goals of the user of the maze. The mazes shown here are simple. In [7] more complex design criteria for mazes are given. In [8], individual evolved maze tiles are used to build scalable maps.

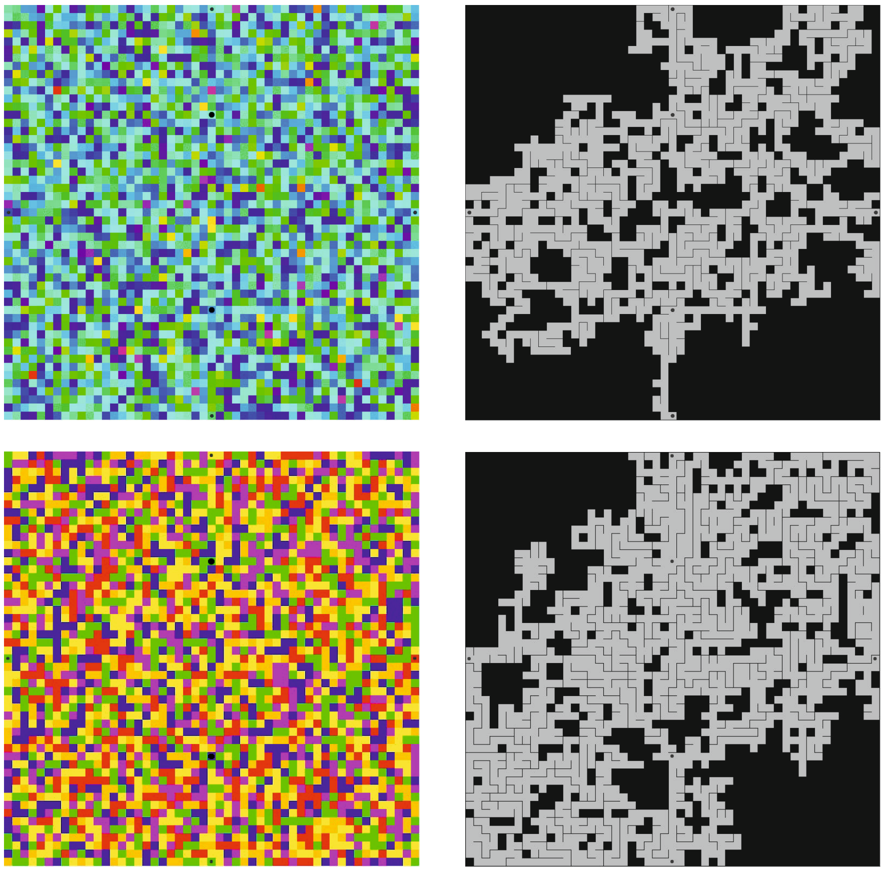

Fig. 9. Examples of height-based and chromatic mazes, together with their keys. The checkpoints are shown as black circles. The minimum height in the height-based mazes is colored red and colors move down the rainbow order of colors as heights increase.

5 Discussion and Conclusions

In all four examples in this chapter it has been demonstrated that representation has a substantial impact on the outcome of an evolutionary algorithm. The example using self avoiding walks showed that changing the representation changed the time to solution, but in different ways for different cases of the problem. This demonstrates that, even within a simple problem domain, the best choice of representation is problem specific. The SAW problem is the simplest system that has, so far, shown this sort of complex response to the change of representation. This makes it a good sandbox for developing tools for exploring the issue of representation.

In the section on evolving agents to play the iterated prisoner's dilemma we saw that the choice of representation can dominate the behavior of a simulation.

This means that a justification of the choice of representation is critical when evolving competing agents. This is the strongest evidence of a need to better understand representation within evolutionary computation. This example goes beyond the issue of performance to that of validity of results.

The examples given in the section on real parameter optimization show that the representation can be chosen to meet particular goals. The Sierpinski representation permits log-time databasing of optima already located and so makes enumeration of optima within the convex hull of its generators a simple matter. The gene expression representation favors exploration over exploitation and so is good for an environment with many optima of differing quality. Both the Sierpinski and gene expression representations are potentially valuable in hybrid algorithms. Each could generate locations near optima that are then finished by a standard hill-climber. Adding a special purpose local optimizer could enhance the performance of each of these representations while permitting them to retain their other special qualities.

The experiments with representations for maps of mazes show that the choice of representation can be used to control the appearance of the output of the algorithm. In the maze evolution project there is no point to comparing the final fitness of the different representations. The needs of a game designer for a particular type of appearance dominate the need to obtain a global best fitness. This speaks to an important point: one should carefully consider one's goals when choosing a representation.

Representation for the mazes is a matter of controlling the appearance of the maze. In real optimization the final goal may be global best fitness, but it might also be obtaining a diverse set of solutions. This latter goal becomes more important if the goal is multi-criteria optimization. With game playing agents, it was demonstrated that representation has a dominant effect on a simple type of experiment. The goal of choice of representation for game playing agents is to simulate some real-world situation. So far we have very little idea of *how* to choose an humaniform or ant-like representation for simulation of conflict and cooperation. This is a wide-open area for future research. The SAW problems are a toy problem that exhibit complex representational effects.

We hope that this chapter has convinced the reader of the importance of considering representation. We conclude by noting that for every example we chose to discuss here we have five others for which there was no room. We invite and appeal to the reader to join in the effort of understanding representation in evolutionary computation.

References

1. Ashlock, D.: Training function stacks to play iterated prisoner's dilemmain. In: Proceedings of the 2006 IEEE Symposium on Computational Intelligence in Games, pp. 111–118 (2006)
2. Ashlock, D., Ashlock, W., Umphry, G.: An exploration of differential utility in iterated prisoner's dilemma. In: Proceedings of the 2005 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, pp. 271–278 (2006)

3. Ashlock, D., Bryden, K.M., Gent, S.: Multiscale feature location with a fractal representation. In: Intelligent Engineering Systems Through Artificial Neural Networks, vol. 19, pp. 173–180 (2009)
4. Ashlock, D., Kim, E.-Y.: Fingerprinting: Automatic analysis and visualization of prisoner's dilemma strategies. IEEE Transaction on Evolutionary Computation 12, 647–659 (2008)
5. Ashlock, D., Kim, E.Y., Leahy, N.: Understanding representational sensitivity in the iterated prisoner's dilemma with fingerprints. Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews 36(4), 464–475 (2006)
6. Ashlock, D., Lee, C., McGuinness, C.: Search-based procedural generation of maze-like levels. IEEE Transactions on Computational Intelligence and AI in Games 3(3), 260–273 (2011)
7. Ashlock, D., Lee, C., McGuinness, C.: Simultaneous dual level creation for games. IEEE Computational Intelligence Magazine 6(2), 26–37 (2011)
8. Ashlock, D., McGuinness, C.: Decomposing the level generation problem with tiles. In: Proceedings of IEEE Congress on Evolutionary Computation, pp. 849–856. IEEE Press, Piscataway (2011)
9. Ashlock, D., Rogers, N.: A model of emotion in the prisoner's dilemma. In: Proceedings of the 2008 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, pp. 272–279 (2008)
10. Ashlock, D.A., Kim, E.Y.: Fingerprint analysis of the noisy prisoner's dilemma. In: Proceedings of the 2007 Congress on Evolutionary Computation, pp. 4073–4080 (2007)
11. Ashlock, D.A., Schonfeld, J.: A fractal representation for real optimization. In: Proceedings of the 2007 Congress on Evolutionary Computation, pp. 87–94 (2007)
12. Ashlock, D., Joenks, M.: ISAc lists, a different representation for program induction. In: Genetic Programming 1998, Proceedings of the Third Annual Genetic Programming Conference, pp. 3–10. Morgan Kaufmann, San Francisco (1998)
13. Axelrod, R.: The Evolution of Cooperation. Basic Books, New York (1984)
14. Beyer, H.-G.: The Theory of Evolution Strategies. Springer, Berlin (2001)
15. Bryden, K.M., Ashlock, D., Corns, S., Willson, S.: Graph based evolutionary algorithms. IEEE Transactions on Evolutionary Computation 5(10), 550–567 (2005)
16. Fogel, L.J.: Intelligence through Simulated Evolution: Forty Years of Evolutionary Programming. John Wiley, Hoboken (1999)
17. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Publishing Company, Inc., Reading (1989)
18. Greene, C.S., Moore, J.H.: Solving complex problems in human genetics using gp: challenges and opportunities. SIGEVOlution 3, 2–8 (2008)
19. Kim, E.Y.: Analysis of Game Playing Agents with Fingerprints. PhD thesis, Iowa State University (2005)
20. Koza, J.R.: Genetic Programming. The MIT Press, Cambridge (1992)
21. McGuinness, C., Ashlock, D.: Incorporating required structure into tiles. In: Proceedings of Conference on Computational Intelligence in Games, pp. 16–23. IEEE Press, Piscataway (2011)
22. Miller, J.H.: The coevolution of automata in the repeated prisoner's dilemma. Journal of Economic Behavior and Organization 29(1), 87–112 (1996)